Java 8 - Streams

Maciej Koziara

Lambda

// Inny sposób zapisu klas anonimowych, krótszy i czytelniejszy
// Nie wymaga ręcznego tworzenia interfejsu
// Nie wymaga podania nazwy metody którą implementuje
// Jeżeli składa się z jednej linijki można pominąć nawiasy klamrowe oraz słowo kluczowe return
// Podawanie typu parametru jest nieobowiązkowe

```
filterApples(apples, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return apple.getColor().equals("red");
    }
});
```

```
filterApples(apples, apple -> apple.getColor().equals("red"));
```

Lambda

// Aby zaaplikować lambdę, interfejs musi posiadać tylko jedną metodę - dzięki temu można jednoznacznie stwierdzić, która metoda powinna zostać użyta
// Lambdy powinny być najkrótsze - najlepiej jedna linijka
// Lambdy nie powinny wprowadzać żadnych efektów ubocznych, tzn. modyfikować zmiennych lub obiektów na których operują

Lambda

// Jeżeli lambda składa się jedynie z wywołania innej metody, można ją zapisać krócej korzystając ze składni **method reference**

```
.map(string -> string.toUpperCase())
.map(volume -> makeJuice(volume))
.map(volume -> new Juice(volume))
```

```
.map(String::toUpperCase)
.map(this::makeJuice)
.map(Juice::new)
```

Stream

// Rozszerzenie interfejsu Collection umożliwiające łatwe przetwarzanie elementów np. listy

// Opiera się na przekazywaniu funkcji (lambd) do metod - dzięki temu określamy **co** powinno się wydarzyć, a nie **jak** to zrobić

Stream

// Posiada dwa rodzaje operacji (metod)
// **intermediate operations** - służą do budowania ciągu poleceń, które później zostaną wykonane na elementach streamu (np. *.filter()*, *.map()*)
// **terminal operations** - operacja kończące. Po ich zastosowaniu zostanie wykonany cały pipeline i zwrócony wynik (np. *collect()*, *count()*)

Stream

// Streamy są **lazy** -> żadna ze zdefiniowanych **intermediate operations** nie zostanie wykonana dopóki nie wykonany operacji kończący (**terminal operation**)
// Dzięki takiemu podejściu streamy mogą wykonać szereg optymalizacji
// Jeżeli Stream zostanie wykonany, nie można go reużyć

Stream - tworzenie

// **list.stream()** - najpopularniejszy sposób. Przekształca daną kolekcję w stream zawierający wszystkie elementy danej kolekcji
// **Stream.of()** - tworzy stream z argumentów przekazanych do metody

Stream - intermediate operations

// **.filter(x -> x.isValid())** - zachowuje, bądź odrzuca element ze streamu zależnie czy metoda zwróci true lub false
// **.map(x -> x.getValue())** - przekształca jeden element na drugi
// **.limit()** - ogranicza ilość elementów w streamie do wskazanej wartości
// **.skip()** - omija n pierwszych elementów w streamie
// **.distinct()** - usuwa duplikaty ze streamu

Stream - sortowanie

// Do sortowania służy metoda **.sorted()**, która przyjmuje jako argument Comparator, który zostanie użyty do sortowania.
// **.sorted()** - sortuje stream zgodnie z sortowaniem naturalnym
// .**sorted(comparing(Person::getName))** - posortuj alfabetycznie, wg. imienia

Stream - ternary operations

// **.collect(Collectors.toList())** - tworzy listę z elementów znajdujących się w streamie
// **.count()** - liczy elementy znajdujące się w streamie
// **.findFirst()** - zwraca pierwszy element w streamie jako **Optional**. Jeżeli stream był pusty, wówczas **Optional** będzie w stanie **empty()**

Stream - ternary operations

// **.anyMatch(i -> i > 10)** - sprawdza czy jakikolwiek element spełnia podany warunek
// **.noneMatch(i -> i > 10)** - sprawdza czy żaden z elementów nie spełnia podanego warunku
// **.allMatch(i -> i > 10)** - sprawdza czy wszystkie elementy w streamie spełniają podany warunek
// **.min(naturalOrder())** - zwraca najmniejszy element (wg. podanego comparatora)
// **.max(reverseOrder())** - zwraca największy element